CSE 4600 Homework 5: Semaphores

Highlights of this assignment:

1. Definition of Semaphore

The original meaning of a semaphore was:

> *"A visual signaling apparatus with flags, lights, or mechanically moving arms, as one used on a railroad." (from http://dictionary.reference.com)*

What do railroad tracks and computers have in common? Well, railroad semaphores were the inspiration of E. W. Dijkstra solution to the mutual exclusion problem in operating systems.

> *"...consider a stretch of railroad in which there is a single track over which only one train at a time is allowed.*
> *Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter."*
> *(from Oracle Documentation: Programming with Synchronization Objects.)*

Railroad semaphores serve as a signal to indicate whether or not a single track has a train on it. Similarly, semaphores in operating systems serve as a signal to indicate whether or not a shared resource is currently being used. The following is a definition of the operating systems semaphores (from The Free On-line Dictionary of Computing, http://www.foldoc.org/, Editor Denis Howe):

> *"[Semaphores are] the classic method for restricting access to shared resources (e.g. storage) in a multi-processing environment. They were invented by Dijkstra and first used in T.H.E operating system.*
>
> *A semaphore is a protected variable (or abstract data type) which can only be accessed using the following operations:*

```
P(s)  /* idea of the sem_wait */
Semaphore s;
{
  while (s == 0) ;      /* wait until s>0 */
  s = s - 1;
}

V(s)    /* idea of the sem_post */
Semaphore s;
{
```

```
          s = s + 1;
        }

        Init(s, v)
        Semaphore s;
        int v;
        {
          s = v;
        }
```

P and V stand for Dutch "Proberen", to test, and "Verhogen", to increment. The value of a semaphore is the number of units of the resource which are free (if there is only one resource a "binary semaphore" with values 0 or 1 is used). The P operation busy-waits (or maybe sleeps) until a resource is available whereupon it immediately claims one. V is the inverse, it simply makes a resource available again after the process has finished using it. Init is only used to initialise the semaphore before any requests are made. The P and V operations must be indivisible, i.e. no other process can access the semaphore during the their execution.

To avoid busy-waiting, a semaphore may have an associated queue of processes (usually a FIFO). If a process does a P on a semaphore which is zero the process is added to the semaphore's queue. When another process increments the semaphore by doing a V and there are tasks on the queue, one is taken off and resumed. "

Note: These operations must be performed as an atomic hardware instruction to prevent two processes from gaining control of the same semaphore. The ideas above are concepts, but only safe when performed through one hardware instruction.

---

2. Why Semaphores

Read the following code `badcnt.c`,

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define NITER 1000000

int cnt = 0;

void * Count(void * a)
{
    int i, tmp;
    for(i = 0; i < NITER; i++)
    {
        cnt++;
    }
}

int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;

    if(pthread_create(&tid1, NULL, Count, NULL))
    {
      printf("\n ERROR creating thread 1");
      exit(1);
    }

    if(pthread_create(&tid2, NULL, Count, NULL))
    {
      printf("\n ERROR creating thread 2");
      exit(1);
    }
```

```
    if(pthread_join(tid1, NULL))        /* wait for the thread 1 to finish */
    {
      printf("\n ERROR joining thread");
      exit(1);
    }

    if(pthread_join(tid2, NULL))        /* wait for the thread 2 to finish */
    {
      printf("\n ERROR joining thread");
      exit(1);
    }

    if (cnt < 2 * NITER)
        printf("\n BOOM! cnt is [%d], should be %d\n", cnt, 2*NITER);
    else
        printf("\n OK! cnt is [%d]\n", cnt);

    pthread_exit(NULL);
}
```

Compile it using

```
gcc badcnt.c -o xbadcnt -lpthread
```

Run the executable file and observe the ouput.

Quite unexpected! Since `cnt` starts at 0, and both threads increment it `NITER` times, we should see `cnt` equal to `2*NITER` at the end of the program. What happens?

Threads can greatly simplify writing elegant and efficient programs. However, there are problems when multiple threads share a common address space, like the variable `cnt` in our earlier example.

To understand what might happen, let us analyze this simple piece of code:

```
THREAD 1                THREAD 2
a = data;               b = data;
a++;                    b++;
data = a;               data = b;
```

Now if this code is executed serially (for instance, `THREAD 1` first and then `THREAD 2`), there are no problems. However threads execute in an arbitrary order, so consider the following situation:

Thread 1	**Thread 2**	**data**
`a = data;`	---	0
`a = a + 1;`	---	0
---	`b = data;  // 0`	0
---	`b = b + 1;`	0
`data = a;  // 1`	---	1
---	`data = b;  // 1`	1

So data could end up 1 or 2, and there is **NO WAY** to know which value! It is completely non-deterministic!

The solution to this is to provide functions that will block a thread if another thread is accessing data that it is using.

Pthreads may use semaphores to achieve this.

---

3. Functions for Posix Semaphores

All POSIX semaphore functions and types are prototyped or defined in `semaphore.h`. To define a semaphore object, use

```
#include <semaphore.h>
sem_t sem_name;
```

sem_init()

sem_init initializes a semaphore.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Notes:

- `sem` points to a semaphore object to initialize
- `pshared` is a flag indicating whether or not the semaphore should be shared with fork()ed processes. Linux Threads does not currently support shared semaphores
- `value` is an initial value to set the semaphore to

Example of use:

```
sem_init(&sem_name, 0, 10);
```

sem_wait()

To wait on a semaphore, use sem_wait:

```
int sem_wait(sem_t *sem);
```

- If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls `sem_post`.

Example of use:

```
sem_wait(&sem_name);
```

sem_post()

To increment the value of a semaphore, use sem_post:

```
int sem_post(sem_t *sem);
```

- It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

Example of use:

```
sem_post(&sem_name);
```

sem_getvalue()

To find out the value of a semaphore, use sem_getvalue:

```
int sem_getvalue(sem_t *sem, int *valp);
```

- gets the current value of sem and places it in the location pointed to by valp.

Example of use:

```
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphors is %d\n", value);
```

sem_destroy()

To destroy a semaphore, use, use sem_destroy:

```
int sem_destroy(sem_t *sem);
```

- destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

```
sem_destroy(&sem_name);
```

Using semaphores - a short example

Consider the problem we had before and now let us use semaphores:

```
Declare the semaphore global (outside of any funcion):
sem_t mutex;

Initialize the semaphore in the main function:
sem_init(&mutex, 0, 1);
```

Thread 1	Thread 2	data
sem wait (&mutex);	---	0
---	sem wait (&mutex);	0
a = data;	/* blocked */	0
a = a + 1;	/* blocked */	0
data = a;	/* blocked */	1
sem post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b + 1;	1
/* blocked */	data = b;	2
/* blocked */	sem post (&mutex);	2
[data is fine. The data race is gone.]		

4. Homework Assignment

Activity 1

Use the example above as a guide to fix the program `badcnt.c`, so that the program always produces the expected output (the value 2*NITER). Make a copy of `badcnt.c` into `goodcnt.c` before you modify the code. To compile a program that uses pthreads and posix semaphores, use

```
gcc -o goodcnt goodcnt.c -lpthread
```

Activity 2

Given the following code,

```
/*
hw5.c
gcc hw5.c -o hw5 -lpthread
*/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define TRUE 1
#define BUFFER_SIZE 5
#define LOOPS 10

/* the semaphores */
sem_t full, empty, mutex;

/* the buffer */
int buffer[BUFFER_SIZE];

/* buffer counter */
int in, out;

pthread_t tid;       //Thread ID
pthread_attr_t attr; //Set of thread attributes

void *producer(void *param); /* the producer thread */
void *consumer(void *param); /* the consumer thread */

void initializeData() {

   /* Create the mutex lock */
   sem_init(&mutex, 0, 1);

   /* Create the full semaphore and initialize to 0 */
   sem_init(&full, 0, 0);

   /* Create the empty semaphore and initialize to BUFFER_SIZE */
   sem_init(&empty, 0, BUFFER_SIZE);

   /* Get the default attributes */
   pthread_attr_init(&attr);

   /* init buffer */
   in = 0;
   out = 0;
}

/* Producer Thread */
void *producer(void *param) {
   int item;
   int i = 0;

   while(i < LOOPS)
   {
      /* generate a random number */
      item = rand()%1000;

      //ADD YOUR CODE
```

```
    }
}

/* Consumer Thread */
void *consumer(void *param) {
    int item;
    int i = 0;

    while(i < LOOPS)
    {

    // ADD YOUR CODE

    }
}

int main(int argc, char *argv[]) {
    /* Loop counter */
    int i;

    /* Verify the correct number of arguments were passed in */
    if(argc != 3) {
        fprintf(stderr, "USAGE:./pc  \n");
    }

    int numProd = atoi(argv[1]); /* Number of producer threads */
    int numCons = atoi(argv[2]); /* Number of consumer threads */

    /* Initialize the app */
    initializeData();

    /* Create the producer threads */
    for(i = 0; i < numProd; i++) {
        /* Create the thread */
        pthread_create(&tid,&attr,producer, NULL);
     }

    /* Create the consumer threads */
    for(i = 0; i < numCons; i++) {
        /* Create the thread */
        pthread_create(&tid,&attr,consumer, NULL);
    }

    /* Sleep for the specified amount of time */
    sleep(1);

    /* Exit the program */
    printf("Exit the program\n");
    exit(0);
}
```

You will be adding code in the producer and consumer functions where the comments indicate to "ADD YOUR CODE".

If you do not know how to start the producer function, you might want to think about the answers to these questions:

- If the "empty" semaphore is 0, what will happen when you perform a sem_wait on the "empty" semaphore? How does the "empty" semaphore change to a positive integer?
- What is the producer doing with the buffer array? Storing values or accessing values?
- What index will you be working with in the producer: *in* or *out*?
- When the producer is done, which semaphore will need to have one added to it (ie. "full" or "empty")?

If you do not know how to start the consumer function, you might want to think about the answers to these questions:

- If the "full" semaphore is 0, what will happen when you perform a sem_wait on the "full" semaphore? How does the "full" semaphore change to a positive integer?
- What is the consumer doing with the buffer array? Storing values or accessing values?
- What index will you be working with in the consumer: *in* or *out*?
- When the consumer is done, which semaphore will need to have one added to it (ie. "full" or "empty")?

Note: these questions do not need to be handed in.

A sample run is the following (yours might be slightly different):

```
Producer [140078164084480] produced 383
Producer [140078164084480] produced 915
  Consumer [140078138906368] consumed 383
Producer [140078155691776] produced 886
Producer [140078155691776] produced 335
  Consumer [140078130513664] consumed 915
Producer [140078164084480] produced 793
  Consumer [140078138906368] consumed 886
Producer [140078155691776] produced 386
Producer [140078147299072] produced 777
  Consumer [140078130513664] consumed 335
  Consumer [140078130513664] consumed 793
Producer [140078164084480] produced 492
  Consumer [140078138906368] consumed 386
  Consumer [140078122120960] consumed 777
Producer [140078147299072] produced 421
Producer [140078147299072] produced 27
  Consumer [140078122120960] consumed 492
  Consumer [140078138906368] consumed 421
Producer [140078155691776] produced 649
Producer [140078155691776] produced 59
  Consumer [140078138906368] consumed 27
  Consumer [140078138906368] consumed 649
Producer [140078155691776] produced 763
Producer [140078164084480] produced 362
  Consumer [140078130513664] consumed 59
  Consumer [140078130513664] consumed 763
Producer [140078147299072] produced 690
Producer [140078147299072] produced 426
  Consumer [140078138906368] consumed 362
  Consumer [140078122120960] consumed 690
  Consumer [140078130513664] consumed 426
Producer [140078164084480] produced 540
Producer [140078155691776] produced 926
Producer [140078155691776] produced 211
Producer [140078147299072] produced 172
  Consumer [140078122120960] consumed 540
  Consumer [140078138906368] consumed 926
  Consumer [140078130513664] consumed 211
Producer [140078147299072] produced 567
Producer [140078155691776] produced 368
  Consumer [140078122120960] consumed 172
  Consumer [140078122120960] consumed 567
Producer [140078147299072] produced 782
  Consumer [140078138906368] consumed 368
........
```

Deliverables:

Submit 3 files to Canvas:

1. *goodcnt.c* with the semaphore version of a mutex
2. producer-consumer code *hw5.c* with the 3 semaphores
3. A script log file *hw5log.txt* showing:
    - compiling *badcnt.c*
    - running of badcnt
    - compiling *goodcnt.c*
    - running of goodcnt
    - compiling producer-consumer *hw5.c* code
    - running of producer-consumer code